Moduł 3: Wstrzykiwanie zależności

W poprzednim module omówiłem sposób tworzenia i konfiguracji kontekstu oraz w jaki sposób deklarować komponenty. Były to bardzo proste przykłady, ale oddawały naturę rzeczy. Posiadając tę podstawową wiedzę można przystąpić do budowania bardziej skomplikowanych aplikacji. Nim jednak do tego przejdziemy, należy omówić w jaki sposób Spring rozwiązuje zależności pomiędzy komponentami.

Istotna uwaga; w kolejnych przykładach całość konfiguracji odbywać się będzie poprzez adnotacje i rola plików konfiguracyjnych zostanie sprowadzona do niezbędnego minimum. Wynika to z faktu, iż nieustanne przenoszenie uwagi pomiędzy kod Javy a pliki konfiguracyjne XML może zaciemniać obraz – szczególnie w początkowej fazie nauki. Tworząc niniejsze szkolenie, wyszedłem z założenia, że najpierw należy poznać zasadę działania i mechanizmy frameworka, a dopiero w kolejnym kroku analizować alternatywne sposoby konfigurowania aplikacji. W jednym z ostatnich rozdziałów zrobię przegląd całego szkolenia i wszystkie konfiguracje automatyczne przepisane zostaną na XML. Wtedy też zostaną podkreślone różnice pomiędzy konfiguracjami oraz omówione ogólne zalecenia kiedy z jakiego podejścia korzystać.

Co więcej, ze względu na wspomniany już początek nauki, wszystkie przykłady nie wykraczały poza podstawowe „Hello World”. Fragmenty programów miały na celu pokazać podstawowe zasady budowania aplikacji, dać posmak różnorodności i elastyczności frameworka. Nie oznacza to oczywiście, że wszystkie przykłady szkolenia będą się ograniczać do trywialnych – już w tym module zacznę pokazywać jak realizować konkretne wymagania biznesowe, aby szkolenie nie było oderwane od rzeczywistości.

Łączenie komponentów

W poprzedniej części analizowaliśmy jak utworzyć komponent, jakie publiczne metody komponentu mogą zostać wywołane przez kontener w momencie inicjacji klasy. Na prostych przykładach pokazane zostało w jak komponenty można ze sobą łączyć.

Wróćmy zatem do wiedzy z poprzedniego modułu i utwórzmy nowy komponent. Aby osadzić nasze rozważania w szerszym kontekście, tworzone przez nas komponenty są elementem większego systemu, który będzie powstawał wraz z rozwojem szkolenia. Będzie to aplikacja do zarządzania zadaniami, czyli tzw. to-do. Prosta (z początku) aplikacja umożliwiająca przeglądanie i dodawanie zadań, układanie ich w pewien mini harmonogram – jest tematyka ogólnie znana, niewymagająca jakiegokolwiek wprowadzenia biznesowego, zatem idealna na przykład.

```
@Component
public  class TasksService {

	List<Task> tasks = new ArrayList<Task>();

	public void addTask(Date startDate,
			int duration,
			String title,
			String description) {

		tasks.add(new Task.Builder()
			.withTitle(title)
			.withDescription(description)
			.withDate(startDate)
			.withDuration(duration)
		.build());

	}

	public List<Task> getTasks() {
		return tasks;
	}

}
```

Jest to niezmiernie naiwna implementacja listy zadań, przechowująca listę w pamięci. Z czasem jednak, będzie ona rozbudowywana, a póki co będzie służyć jako baza do dalszych rozważań. Należy tutaj przypomnieć o czym pisałem w poprzednim module: każdy serwis zarządzany przez Spring jest singletonem, tzn. istnieje tylko jedna instancja danego komponentu w kontekście. Oznacza to że wszystkie serwisy korzystające z komponentu TaskService będą miały dostęp do tej samej listy zadań. W ogólnym przypadku (np. wielowątkowej aplikacji webowej) byłby to niewybaczalny błąd, jednakże w przykładowej aplikacji możemy wykorzystać tę właściwość, umożliwiając komponentowi przechowywanie danych.

Sam serwis dodający zadania to jednak mało; projektując aplikację najbardziej zależy nam na możliwości przeglądania dodanych zadań – potrzeba do tego metody wyświetlającej zadania, np. na ekran. Podskórnie jednak czujemy, że dodanie takiej metody do serwisu zajmującego się dodawaniem zadań nie jest najlepszym pomysłem. Dobrze zaprojektowana aplikacja obiektowa ma jednoznacznie rozdzielone odpowiedzialności; każda klasa powinna realizować funkcjonalności zbliżone biznesowo, a dodawanie zadań i wyświetlanie ich na ekran nie koniecznie są powiązane w aż takim stopniu. Do wyświetlania na ekranie tworzymy kolejny serwis

```
@Component
public  class TaskPrinter {

	@Autowired
	TasksService tasksService;

	public void printCurrentTasks() {
		List<Task> tasks = tasksService.getTasks();
		for (Task t : tasks) {
			System.out.println(t);
		}
	}

}
```

Do powiązania dwóch komponentów użyliśmy adnotacji `@Autowired` (o której wspomniane zostało już w poprzednim module). W naszym przypadku adnotacją oznaczyliśmy pole klasy, chociaż równie dobrze powiązanie komponentów może się odbywać poprzez metodę modyfikatora (setter) albo

konstruktor. Tak jak widać na poniższych przykładach, wszystkie konfiguracje są równoważne i dają taki sam efekt.

```
@Component
public  class TaskPrinter {

	private TasksService tasksService;

	public TasksService getTasksService() {
		return tasksService;
	}

	@Autowired
	public void setTasksService(TasksService tasksService) {
		this.tasksService = tasksService;
	}

	//..
}
```

```
@Component
public  class TaskPrinter {

	private TasksService tasksService;

	@Autowired
	public TaskPrinter(TasksService s) {
		this.tasksService = s;
	}

	//..
}
```

Co jednak należy pamiętać, to aby przestrzegać jednej konwencji konfiguracyjnej i unikać mieszania różnych podejść w jednej aplikacji: np. wstrzykiwania poprzez konstruktor w niektórych klasach i poprzez metodę w innych. Wprowadza to tylko niepotrzebne zamieszanie w kodzie. Konwencją przyjętą w niniejszym szkoleniu jest wstrzykiwanie zależności poprzez pole klasy, co wynika tylko i wyłączenie z przyzwyczajeń autora (taka sama metoda stosowana jest w aplikacjach JEE opartych o komponenty EJB – stąd też próba ujednolicenia stylu).

Adnotacja `@Autowired` posiada tylko jeden parametr, będący typu *boolean*: *required*. Domyślna wartość to *true*, która wymaga aby zależności były spełnione w momencie inicjalizacji kontenera. Jeżeli nastąpi próba wstrzyknięcia nieistniejącego komponentu, w momencie inicjalizacji aplikacji zwrócony zostanie wyjątek `NoSuchBeanDefinition`.

Alternatywnie, ustawiając wartości flagi na *false*, kontener staje się mniej restrykcyjny i nie zgłasza błędu w sytuacji gdzie wstrzykiwany komponent nie istnieje w kontekście. Jeżeli nastąpi próba odwołania się do takiego pola, zwrócony zostanie standardowy wyjątek: `NullPointerException`. Sytuacje, w których decydujemy się na opcjonalność komponentu są niezmiernie rzadkie i teraz nie będziemy się nimi zajmować.

Więcej niż jedna instancja

Nie zastanawialiśmy się dotychczas co się dzieje gdy w kontekście istnieje więcej niż jedna instancja komponentu (np. kilka komponentów implementuje ten sam interfejs). W takich sytuacjach sama adnotacja `@Autowire` okazuje się być niewystarczająca. Powyższy przykład był niezwykle prosty, spróbujmy go zatem rozszerzyć o kolejny serwis, wypisujący zadania w nieco innej formie. W

pierwszej kolejności wyodrębnimy interfejs TaskPrinter oraz zmienimy nazwę istniejącej klasy na ToStringTaskPrinter

```
public  interface TaskPrinter {

	public void printCurrentTasks();

}
```

```
@Component
public  class ToStringTaskPrinter implements TaskPrinter {

	@Autowired
	TasksService tasksService;

	public void printCurrentTasks() {
		List<Task> tasks = tasksService.getTasks();
		for (Task t : tasks) {
			System.out.println(t);
		}
	}

}
```

Wciąż odwołania do nazwanego komponentu odbywać się mogą poprzez `@Autowire`, ponieważ nadal w kontekście istnieje tylko jeden komponenty typie TasksPrinter. Działanie aplikacji nie zmieniło się.

```
	@Autowired
	TaskPrinter printer;

	public void print() {
		printer.printCurrentTasks();

	}
```

Utwórzmy zatem dodatkowy komponent, wyświetlający zadania w zmienionej formie.

```
@Component
public  class PrettyStringTaskPrinter implements TaskPrinter {

	@Autowired
	TasksService tasksService;

	public void printCurrentTasks() {
		List<Task> tasks = tasksService.getTasks();
		for (Task t : tasks) {
			StringBuilder b = new StringBuilder();
			b.append(" * ").append(t.getTitle())
				.append(" (").append(t.getDescription()).append(")\n")
				.append("\t").append("[")
					.append(t.getDate()).append(" for ")
					.append(t.getDuration()/60/60)
					.append(" hours")
				.append("]");
			System.out.println(b.toString());
		}
	}

}
```

Kolejna próba uruchomiania aplikacji skończy się niepowodzeniem, ponieważ istnieje więcej niż jeden komponent implementujący ten sam interfejs (toStringTasksPrinter oraz

prettyStringTasksPrinter). Istnieje kilka sposobów naprawienia tej sytuacji, od zmiany nazewnictwa (i zastosowanie pewnych konwencji) na wykorzystaniu innych adnotacji skończywszy.

Zmiana nazwy pola

Jak już sygnalizowałem wcześniej, bardzo wiele rzeczy w Springu opartych jest o konwencje nazewnicze. Dzięki temu, odpowiednio nazywając pola, klasy, można oszczędzić bardzo dużo kodu konfiguracyjnego (tzw. *boiler plate code*) i skupić się na implementowaniu oraz realizacji wymagań. Jedną z takich konwencji możemy zastosować tutaj, zmieniając nazwę pola z printer na nazwę odpowiadającą nazwie komponentu który chcemy wstrzyknąć.

```
	@Autowired
	TaskPrinter toStringTaskPrinter;

	public void print() {
		toStringTaskPrinter.printCurrentTasks();

	}
```

```
	@Autowired
	TaskPrinter prettyStringTaskPrinter;

	public void print() {
		prettyStringTaskPrinter.printCurrentTasks();

	}
```

Framework samodzielnie próbuje rozwiązać wieloznaczność poprzez sprawdzenie czy nazwa pola pasuje do nazwy komponentu w kontekście. Zdaję sobie sprawę że nie jest to najbardziej elegancka metoda i jeżeli komuś przeszkadza pewna doza magii w aplikacji, to alternatywą użycie adnotacji `@Qualifier`, która jednoznacznie wskazuje implementację do wstrzyknięcia.

Dodatkowa adnotacja @Qualifier

Adnotacją `@Qualifier` jest swego rodzaju wzorcem, który możemy dowolnie parametryzować aby dopasować zależności do własnych potrzeb. Teoretycznie wytłumaczenie zasady działania tej adnotacji jest dość zawoalowane, posłużmy się więc przykładem.

Wstrzykiwany komponent można oznaczyć adnotacją `@Qualifier`, a następnie posłużyć się zadeklarowaną wartością podczas konfigurowania zależności:

```
	@Component
	@Qualifier ("pretty")
	public  class PrettyStringTaskPrinter implements TaskPrinter {
		//..
	}
```

```
	@Autowired
	@Qualifier("pretty")
	TaskPrinter p;

	public void print() {
		p.printCurrentTasks();
	}
```

Jest to rozwiązanie trochę bardziej jawne niż bazowanie na samych nazwach atrybutów klasy, wciąż jednak trudne do utrzymania, ze względu na obecność ciągów znaków, których refaktoring jest niezmiernie skomplikowany i pełen pułapek.

Możemy zatem utworzyć własny kwalifikator, który następnie będzie wykorzystywany w aplikacji.

```
public enum Printers {
	TO_STRING, PRETTIFY;
}
```

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention (RetentionPolicy.RUNTIME)
@Qualifier
public  @interface PrintQualifier {

	Printers printerType();

}
```

W kolejnym kroku, używamy utworzonego kwalifikatora do rozróżnienia komponentów które pragniemy wstrzykiwać oraz do samego procesu wstrzykiwania zależności.

```
@Component
@PrintQualifier (printerType = Printers.PRETTIFY)
public  class PrettyStringTaskPrinter implements TaskPrinter {
	//..
}
```

```
	@Autowired
	@PrintQualifier(printerType=Printers.PRETTIFY)
	TaskPrinter p;

	public void print() {
		p.printCurrentTasks();
	}
```

Wstrzykiwanie więcej niż jednej instancji

Ostatnim sposobem, dającym największą elastyczność, jest wstrzyknięcie wszystkich możliwych instancji danego typu. W tym celu wykorzystana zostanie funkcjonalność frameworka, pozwalająca na automatyczne tworzenie list (i tablic) na podstawie komponentów kontekstu. Posłużmy się przykładem:

```
@Autowired
List<TaskPrinter> printers;

public void print() {
	for (TaskPrinter p : printers)
		p.printCurrentTasks();
}
```

Powyższa konstrukcja powoduje, że Spring wstrzyknie wszystkie możliwie komponenty pasujące do interfejsu TaskPrinter i rolą programisty będzie wybór najodpowiedniejszego (lub wszystkich – jak na przykładzie powyżej). Przykład ten może się wydawać dość abstrakcyjny (podobnie jak parametr *required* adnotacji `@Autowired`, ustawiony na *false*) jednak ma swoje bardzo konkretne uzasadnienie.

Połączenie obu wspomnianych konstrukcji umożliwia bardzo wygodną budowę rozszerzalnych aplikacji (wspierających pluginy). W sytuacji gdy umożliwimy wstrzyknięcie kolekcji komponentów (a nie tylko konkretnej implementacji) oraz oznaczymy tę zależność jako opcjonalną, uzyskujemy bardzo wygodny sposób budowania wtyczek do naszej aplikacji. Jako że Spring może skanować cały classpath (nie tylko w obrębie jednego pliku jar), możemy łączyć kilka modułów, umożliwiając wybór pomiędzy różnymi implementacjami pewnych funkcjonalności. Wybór dokonywany jest przez programistę w oparciu o np. dodatkowe opcje konfiguracyjne bądź specyficzne własności serwera na którym aplikacja jest wdrażana.

Wstrzykiwanie zasobów

Analogicznie do tworzonych przez nas komponentów, możemy wstrzykiwać zasoby dostarczane przez serwer, kontener, system operacyjny. Jednakże w tym celu należy skorzystać z adnotacji `@Resource`. W odróżnieniu od adnotacji `@Autowire`, `@Resource` nie jest zdefiniowana przez framework, a jest elementem języka Java (JSR-250). Jest to standardowy element języka i poprzez to jej użycie jest w pewnym sensie ograniczone do pól i metod, nie można za jej pomocą oznaczać konstruktora klasy.

Takim zasobem wstrzykiwanym przez `@Resource` może być klasa `ApplicationContext` lub `BeanFactory`, czyli związanych ze środowiskiem w którym działa tworzona aplikacja. Jeżeli jest to aplikacja webowa, może być to `WebApplicationContext`, który jest w stanie dostarczyć szeregu dodatkowych informacji – jak np. bazowy URL serwisu.

Zasadniczo rzecz biorąc, adnotacje `@Autowire` i `@Resource` mogą być używane zamiennie, z tym że działają one w nieco odmienny sposób. Jeżeli pole oznaczone jest adnotacją `@Autowire`, to Spring przede wszystkim poszukuje klas o odpowiednim typie (tożsamym z typem, który został oznaczony). W przypadku `@Resource` sprawdzana jest przede wszystkim nazwa atrybutu (a w kolejnym kroku wartość atrybuty *name* adnotacji). Poszukiwane są klasy o nazwie odpowiadającej polu, a gdy te nie są znalezione – dopiero następuje dopasowanie po typie.

Przytoczony wyżej przykład wieloznaczności może być zatem także rozwiązany za pomocą adnotacji `@Resource`, gdzie poprzez atrybut *name* podajemy nazwę komponentu.

```
@Resource(name = "prettyStringTaskPrinter")
TaskPrinter p;

public void print() {
	p.printCurrentTasks();

}
```

Nie zmienia to jednak faktu, że najlepszy i najbardziej eleganckim rozwiązaniem jest wstrzyknięcie jednej z wielu możliwych implementacji poprzez użycie kwalifikatorów, czyli adnotacji `@Qualifier`.

Usuwane są w ten sposób wszystkie, potencjalne niejasności i jest to najbardziej jednoznaczny sposób oznaczania klas.

Inicjacja komponentów

Nie zastanawialiśmy się jednak nad parametrami, które mogą towarzyszyć tworzeniu obiektu; w jaki sposób utworzyć komponent ze zdefiniowanymi wartościami domyślnymi. Może się jednak zdarzyć, że pewne komponenty pragniemy inicjować dodatkowymi danymi – na zasadzie domyślnych wartości. Używamy do tego adnotacji `@Value`, którą oznaczamy pole klasy i jako argument przekazujemy inicjalną wartość. Jeżeli chcielibyśmy aby komponent wyświetlający wartość zadania na ekran (klasa `PrettyStringTaskPrinter`) wyświetlała datę w sposób sformatowany możemy posłużyć się właśnie adnotacją `@Value`.

```
@Component
@PrintQualifier (printerType = Printers.PRETTIFY)
public  class PrettyStringTaskPrinter implements TaskPrinter {

	@Value("MM/dd/yyyy")
	String pattern;

	@Autowired
	TasksService tasksService;

	//..

}
```

W ten sposób domyślnego formatowania daty nie zapisujemy na sztywno w kodzie aplikacji, ale umożliwiamy jego swobodną zmianę (np. z poziomu kontenera). Można jednak pójść krok dalej: zdefiniować plik konfiguracyjny (plik date-format.properties) który będzie zawierał dodatkowe informacje na temat formatowania daty – i klucz z tego pliku możemy przywołać za pomocą adnotacji `@Value`. Dodatkowo, należy wskazać kontenerowi lokalizację tego pliku, co odbywa się także przez plik konfiguracyjny XML (app-config.xml), w którym dopisujemy następującą linijkę:

```
<context:property-placeholder location="classpath:date-format.properties" />
```

Sam plik z właściwościami jest niezmiernie prosty:

```
	date.pattern = MM/dd/yyyy
```

Tak przygotowaną konfigurację możemy już bezpośrednio wykorzystać w aplikacji:

```
@Component
@PrintQualifier (printerType = Printers.PRETTIFY)
public  class PrettyStringTaskPrinter implements TaskPrinter {

	@Value("${date.pattern}")
	String pattern;

	@Autowired
	TasksService tasksService;

	//..

}
```

Podsumowanie

W poprzednich dwóch modułach stosunkowo dokładnie omówiłem w jaki sposób rozwiązywane są zależności pomiędzy komponentami i w jakiś sposób programista może mieć wpływ na realizację tych powiązań. W kolejnym module wykorzystamy tą wiedzę aby połączyć aplikację z bazą danych (wstrzyknąć komponenty związane z obsługą bazy) i zastąpić przechowywanie zadań wewnątrz komponentu zadaniami zapisywanymi w sposób trwały w bazie danych.